Advances in Programming Languages

APL15: Concurrency abstractions

David Aspinall

School of Informatics
The University of Edinburgh

Monday 2 March 2009
Semester 2 Week 8

Techniques for concurrency

This is the second of three lectures presenting some programming-language techniques for managing concurrency.

- Introduction, basic Java concurrency
- Concurrency abstractions
- Concurrency in some other languages

Outline

Managing synchronization

How do we manage synchronization across many objects in a system?

A *synchronization policy* is necessary to describe which locks are used to protect which pieces of shared data, in which order they are obtained. The programmer should explain which classes are considered *thread safe*, especially for library classes.

Informally, a *thread safe* class is a class whose methods may be invoked from different threads at the same time. A precise definition is *much* more tricky.

- a good idea to document this, e.g. with an *annotation* `@ThreadSafe`.

Managing synchronization

How do we manage synchronization across many objects in a system?

A *synchronization policy* is necessary to describe which locks are used to protect which pieces of shared data, in which order they are obtained. The programmer should explain which classes are considered *thread safe*, especially for library classes.

Informally, a *thread safe* class is a class whose methods may be invoked from different threads at the same time. A precise definition is *much* more tricky.

- a good idea to document this, e.g. with an *annotation* `@ThreadSafe`.

A very naive approach to concurrency problems is to fix concurrency bugs by successively adding more uses of synchronize. Just like deleting statements that cause runtime errors, this rarely succeeds (why not?).

Thread safe collections

Some basic Java collection classes are not thread safe, but convenient wrapper methods can add synchronization around accessor methods.

```
List<Customer> customerList =
      Collections.synchronizedList(new ArrayList<Customer>());

addCustomers(customerList);

for (Customer c : customerList) {
    processCustomer(c);
}
```

Despite using a synchronized list, it is still possible for this code to throw ConcurrentModificationException. Why? How could it be avoided?

Thread safe collections

Some basic Java collection classes are not thread safe, but convenient wrapper methods can add synchronization around accessor methods.

```
List<Customer> customerList =
      Collections.synchronizedList(new ArrayList<Customer>());

addCustomers(customerList);

for (Customer c : customerList) {
    processCustomer(c);
}
```

The addCustomers call *leaks* the reference to the customer list. It's possible that another thread retains this and manipulates the list after return.

Thread safe collections

Some basic Java collection classes are not thread safe, but convenient wrapper methods can add synchronization around accessor methods.

```
List<Customer> customerList =
      Collections.synchronizedList(new ArrayList<Customer>());

addCustomers(customerList);

for (Customer c : customerList) {
    processCustomer(c);
}
```

The implicit iteration in the **for** loop may interact with another thread that is modifying the list. We must lock the whole list while iterating.

Thread safe collections

Some basic Java collection classes are not thread safe, but convenient wrapper methods can add synchronization around accessor methods.

```
List<Customer> customerList =
      Collections.synchronizedList(new ArrayList<Customer>());

addCustomers(customerList);

for (Customer c : customerList) {
    processCustomer(c);
}
```

Confusingly, even single-threaded code can throw ConcurrentModificationException, when the API contract of call sequences is violated by modifying during iteration; a *fail fast* policy is to detect this between calls and abort.

Concurrent collections

The drawback with synchronized compound objects is that further locking may be required when executing compound operations, so we haven't automatically solved consistency problems.

More crucially, they may become a *serialization bottleneck* as serialising accesses prevents concurrency.

The *concurrent collections* introduced in Java 5.0 allow a remedy.

Interfaces:

- Queue
- List
- ConcurrentMap
- BlockingQueue

Classes:

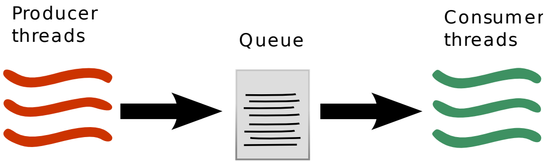
- ConcurrentLinkedQueue
- CopyOnWriteArrayList
- ConcurrentHashMap
- ArrayBlockingQueue

These live in the package java.util.concurrent alongside other utilities. They use various mechanisms to give thread safe results, including *non-blocking* algorithms and lower-level features such as *atomic variables*.

Queues and producer-consumer patterns

The producer-consumer pattern is a common way to decouple jobs and achieve scalable parallelism.

A queue acts as thread-safe "glue" which allows independent tasks to proceed on each side without interfering. Consumers block when the queue is empty; producers block when the queue full.



BlockingQueue access methods

	Throws ex'n	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	–	–

Outline

Application frameworks

Application frameworks separate duties and isolate subparts of a system by using different threads for different tasks. For example:

- The Java Virtual Machine runs a thread for executing the program's main() method, which may start further threads; it also runs *daemon* threads for housekeeping tasks such as garbage collection.
- Swing applications create a GUI thread which uses an input event queue; all GUI operations are confined to the GUI thread.
- J2EE application servers use a *thread pool* to use for container tasks.

To manage finer grained concurrency in a system, some form of additional management on top of threads is often desirable, for example, to manage *work queues* and *tasks* effectively.

Finer granularity allows applications to avoid excessive overhead, by reducing the amount of context switching and the load on a single system level scheduler (which may have hard limits).

Tasks and Executors

Effective concurrent programs subdivide work into *tasks*, which are as independent as possible. Some types of tasks may have to be executed in a given sequence, one at a time. Others may be executed concurrently in multiple threads.

Java provides *executors* as an abstraction for work queues which execute tasks. An executor usually encapsulates one or more threads.

```
public interface Runnable {
  void run();
}

public interface Executor {
  // execute command at some time
  void execute(Runnable command);
}
```

```
Executor workerExecutor =
        Executors.newFixedThreadPool(5);

// schedule 20 jobs immediately
for (int i = 0; i<20; i++) {
   WorkerJob job = new WorkerJob();
   workerExecutor.execute(job);
}
```

Futures

Simple Runnable tasks do something and then finish. To communicate a result, we use *futures*, which are an abstraction of asynchronous result-returning computations. Futures can also be managed by executors.

```
public interface Callable<V> {
  V call() throws Exception;
}

public interface Future<V> {
  V get();
// maybe blocking
  void cancel();
  boolean isCancelled();
  boolean isDone();
}
```

```
FutureTask<Integer> searchFuture =
new FutureTask<String>(new Callable<String>() {
  public String call() {
    return searcher.findMatch(target);
}});

// search while we do something else
executor.execute(searchFuture);
...
if (searchFuture.isDone()) {
  result = searchFuture.get();
} else {
  result = "not found";
  searchFuture.cancel();
}
```

Summary

Java concurrency abstractions

Java provides concurrency extensions in the library java.util.concurrent. These include:

- *concurrent collections* which include scalable thread-safe classes for lists, maps and queues;
- *task management* at a finer granularity than threads, using executors which provide a range of thread pooling and (simple) scheduling strategies.
  - Java 7 will have a *fork-join* library for managing tasks which subdivide themselves, and executors which schedule tasks with *work stealing*.
- *futures* which are asynchronous tasks that return results.

Similar abstractions are available in other languages and libraries.